Prime Numbers in RSA Cryptosystem

Sam Ream and Weimo Zhu, mentored by Charles Kulick

University of California, Santa Barbara - Directed Reading Program (DRP) 2024

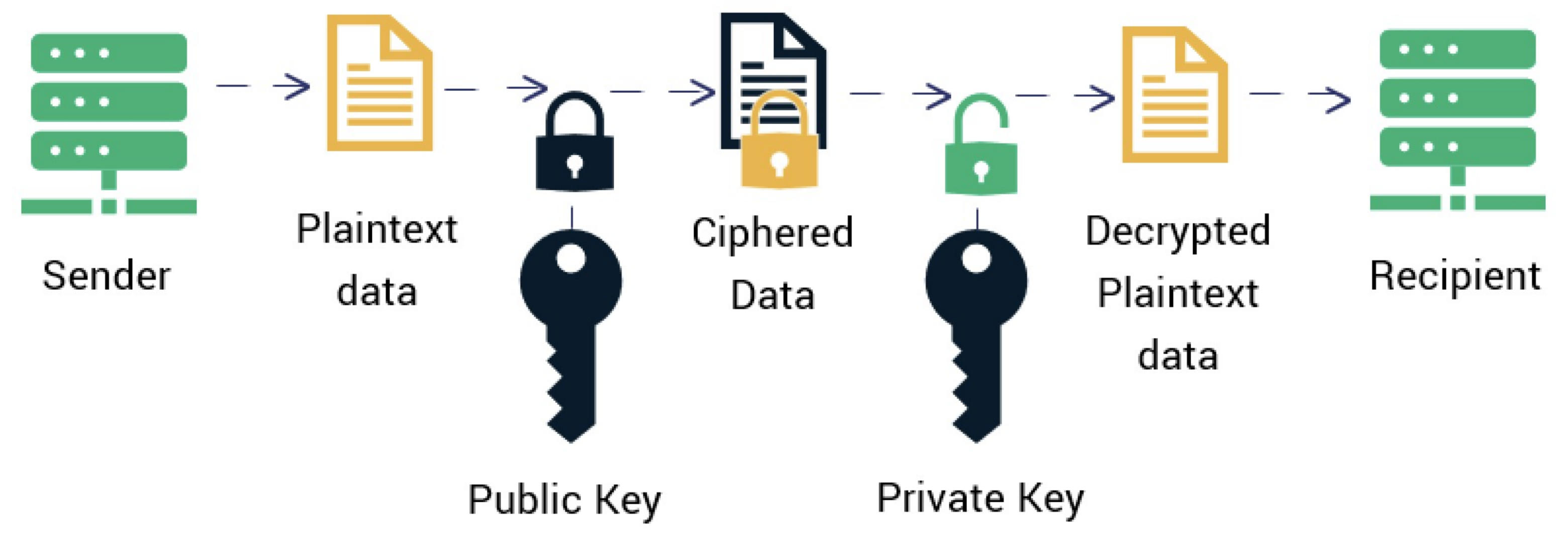
Introduction

Throughout history, the need for secure communication has always been an important issue in various areas, such as private communication during war or credit card encryption. Prior to 1970s, symmetric-key cryptosystems were mainly implemented. Such cryptosystems required the sender and the receiver to agree on a private key, which led to the difficulty of finding a secure line and exchanging keys without being intercepted. Later, public-key cryptosystems (asymmetric cryptosystems) were invented, where the sender and receiver could publicly agree on the public key and set their own private keys. Without the need to send private keys, public-key cryptosystems are much less vulnerable. The RSA cryptosystem is one of the most famous public-key cryptosystems.

RSA Algorithm

1. Receiver: Choose two distinct large prime numbers p and q.
2. Receiver: Compute $n = pq$ and $\phi(n) = (p-1)(q-1)$.
3. Receiver: Choose an integer e such that $gcd(e, \phi(n)) = 1$.
4. Receiver: Compute d such that $de \equiv 1 \pmod{\phi(n)}$.
5. Receiver: Send n, e publicly.
6. Sender: Send $c \equiv m^e \pmod{n}$.
7. Receiver: Decrypt $m \equiv c^d \pmod{n}$.

In general, d is the private key, (n, e) are the public keys.



Significance of Prime Numbers

To attack RSA, the most straightforward way is to factor n. With $n = pq$, the observer can compute $\phi(n) = (p-1)(q-1)$, and thus get the private key d. In modern implementations of the RSA cipher, the prime numbers p and q chosen to compute the encryption key n should be at least hundreds of digits long to ensure security. However, up until now, there was no known factoring method for the product of two hundreds-digit-length primes that could be done in a feasible time frame. Therefore, the high security level of RSA cryptosystems relies on the difficulty of factoring n to get p and q. In the following sections, we will introduce methods to find large primes needed for decryption and also possible attacks on factoring the product of two large primes.

References

[1] Evan Chen.
An Infinitely Large Napkin.
Evan Chen, 2023.

[2] Simon Rubinstein-Salzedo.
Cryptography.
Springer, 2018.

[3] Lawrence C. Washington Wade Trappe.
Introduction to Cryptography with Coding Theory.
Pearson, 2006.

Primality Testing

To choose our p and q, we need to find large, unique numbers and ensure their primality. Instead of trying to factorize an integer x, we can use primality testing to be more efficient.

In the following, we will introduce two probabilistic primality tests, which determine how likely it is that a given integer is prime.

Fermat Primality Test

Recall Fermat's Little Theorem:
If p is a prime number and a is not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$.

Implementing the contrapositive of Fermat's Little Theorem, Fermat "Primality Test" is actually a "Composite Test". It concludes x is composite if there exists an integer a such that $gcd(x, a) = 1$ and $a^{x-1} \not\equiv 1 \pmod{x}$. While if $a^{x-1} \equiv 1 \pmod{x}$, x is probably prime.

Given that this is a probabilistic primality test, there are infinitely many cases where this test fails. They are called Carmichael numbers or absolute pseudoprimes. For example, the smallest Carmichael number is 561.

Miller-Rabin Primality Test

Compared to Fermat Primality Test, Miller-Rabin Primality Test is stronger and has a lower probability in concluding a composite number as prime. It is currently used in many RSA implementations.

The main idea:

1. Given n, an odd integer.
2. Write $n - 1 = 2^k m$, where m is odd now.
3. Choose a random positive integer a such that $1 < a < n - 1$.
4. Compute $b_0 \equiv a^m \pmod{n}$.
5. If $b_0 \equiv \pm 1 \pmod{n}$, then stop the test and n is probably prime.
   If not, continue the test and compute $b_i \equiv b_{i-1}^2 \pmod{n}$.
6. If $b_i \equiv 1 \pmod{n}$, then n is composite.
   If $b_i \equiv -1 \pmod{n}$, then n is probably prime.
7. Iterate until stopping or reaching b_{k-1}.
   If $b_{k-1} \not\equiv -1 \pmod{n}$, then n is composite.
   If not, then n is probably prime.

Scan this QR code for the Python implementation of Miller-Rabin Primality Test.

For a given integer n and a choice of a, the probability of Miller-Rabin Test failing and wrongly declaring that a composite n is prime is at most $\frac{1}{4}$. Thus, if this test is repeated for k times, the probability of failing is at most $(\frac{1}{4})^k$. Repeating 5 times with 5 different a, we can reduce the probability of error to below 0.1%, which is usually accurate enough.

Reconsider the Carmichael number $n = 561$. Then $n - 1 = 560 = 2^4 \times 35$. Let $a = 2$. Then:

$$b_0 \equiv 2^{35} \equiv 263 \pmod{561}$$
$$b_1 \equiv {b_0}^2 \equiv 166 \pmod{561}$$
$$b_2 \equiv {b_1}^2 \equiv 67 \pmod{561}$$
$$b_3 \equiv {b_2}^2 \equiv 1 \pmod{561}$$

Since $b_3 \equiv 1 \pmod{561}$, 561 is correctly declared to be composite.

Factoring Attacks

The Birthday Attack

The motivation behind the birthday attack is the idea that, for example, if there are 23 people in a room, then there is about a 50% chance of two people sharing a birthday and additionally that probability increases to about 70% with 30 people in the room. In general, if there were n unique birthdays, it would take about $\sqrt{2n\log(2)}$ people before we would expect a match.

The birthday attack uses this idea to find factors of n. It will take us about $\sqrt{2n\log(2)}$ random numbers before we find a factor of n. The following algorithm allows us to do this efficiently:

1. Choose a random polynomial f(x) that maps values $\mathbf{Z}/n\mathbf{Z} \Rightarrow \mathbf{Z}/n\mathbf{Z}$.
2. Let $x = 2$ and $y = 2$ (standard convention).
3. Replace x with f(x) and y with f(f(y)).
4. Compute $d = gcd(|x - y|, n)$.
5. If $d = 1$, return to step 3. If $d = n$, then the algorithm fails, so we must restart at step 1 and pick a new function f(x). Otherwise, if $d \neq 1$ and $d \neq n$, then d is our factor of n.

Quadratic Sieve

In this factoring method, if we want to factor some number n, we must find integers x and y such that $x^2 \equiv y^2 (mod\, n)$, but $x \not\equiv y (mod\, n)$. In this case, n is composite and gcd(x-y,n) gives us our nontrivial factor of n.

In order to find our integers x and y, we must produce squares that are slightly larger than a multiple of n using $\lfloor in + j \rfloor$ for various values of i and small j.

Once we find our x integers, we must write them as products of primes less than 20, which will comprise our factor base. Each of our squares will represent a row of a matrix with the entries being the exponents of the primes. For example:

$$\begin{array}{r|cccccccc}
9398 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 1 \\
19095 & 2 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1964 & 0 & 2 & 0 & 0 & 0 & 3 & 0 & 0 \\
17078 & 6 & 2 & 0 & 0 & 1 & 0 & 0 & 0 \\
8077 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
3397 & 5 & 0 & 1 & 0 & 0 & 2 & 0 & 0 \\
14262 & 0 & 0 & 2 & 2 & 0 & 1 & 0 & 0
\end{array}$$

Now, if we have a linear dependency mod 2 among the rows, the product of the numbers yields another square, our y^2. If $x \not\equiv \pm y (mod\, n)$, then gcd(x-y,n) gives us our factor of n.

The $p-1$ Factoring Algorithm

1. Choose an integer $a > 1$ ($a = 2$ is common).
2. Choose a bound B. The size of B will depend on the situation, but if B is too small, the chance of success is small and if B is too big, then the algorithm is very slow.
3. Compute $b \equiv a^{B!} (mod\, n)$ where
   a. $b_1 \equiv gcd(b - 1, n)$.
   b. $b_j \equiv b_{j-1}^j (mod\, n)$.
   c. Then $b_B \equiv (mod\, n)$.
4. Let $d = gcd(b - 1, n)$. If $1 < d < n$, we have our nontrivial factor of n.